Halting problem undecidability and infinitely nested simulation (V2)

The halting theorem counter-examples present infinitely nested simulation (non-halting) behavior to every simulating halt decider. This paper has been rewritten to be more compelling and more concise.

The pathological self-reference of the conventional halting problem proof counter-examples is overcome. The halt status of these examples is correctly determined. A simulating halt decider remains in pure simulation mode until after it determines that its input will never reach its final state. This eliminates the conventional feedback loop where the behavior of the halt decider effects the behavior of its input.

The x86utm operating system was created so that the halting problem could be examined concretely in the high level language of C. H is a function written in C that analyzes the x86 machine language execution trace of other functions written in C. H recognizes simple cases of infinite recursion and infinite loops. The conventional halting problem proof counter-example template is shown to simply be an input that does not halt.

H simulates its input with an x86 emulator until it determines that its input would never halt. As soon as H recognizes that its input would never halt it stops simulating this input and returns 0. For inputs that do halt H acts exactly as if it was an x86 emulator and simply runs its input to completion and then returns 1.

> In theoretical computer science the random-access stored-program (RASP) machine model is an abstract machine used for the purposes of algorithm development and algorithm complexity theory. ...The RASP is closest of all the abstract models to the common notion of computer.
> https://en.wikipedia.org/wiki/Random-access_stored-program_machine

The C/x86 model of computation is known to be Turing equivalent on the basis that it maps to the RASP model for all computations having all of the memory that they need. As long as an C/x86 function is a pure function of its inputs the C/x86 model of computation can be relied upon as a much higher level of abstraction of the behavior of actual Turing machines.

This criteria merely relies on the fact that the UTM simulation of a machine description of a machine is computationally equivalent to the direct execution of this same machine:

Simulating Halt Decider Theorem (Olcott 2021)(V3):
Whenever simulating halt decider H correctly determines that it simulation of input P would never reach its final state (whether or not this simulation is aborted) then H correctly decides that P never halts.

> In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. https://en.wikipedia.org/wiki/Halting_problem

As long as the halt decider H reports the halt status of the behavior of its actual input P then H is necessarily correct no matter how P behaves in any other situation.

Because H only acts as a pure simulator of its input until after its halt status decision has been made it has no behavior that can possibly effect the behavior of its input. Because of this H screens out its own address range in every execution trace that it examines. This is why we never see any instructions of H in any execution trace after an input calls H.

Pathological Input to a halt decider is stipulated to mean any input that was defined to do the opposite of whatever its corresponding halt decider decides as Sipser describes:

> Now we construct a new Turing machine D with H as a subroutine.
> This new TM calls H to determine what M does when the input to M
> is its own description ⟨M⟩. Once D has determined this information,
> it does the opposite. (Sipser:1997:165)

When D is invoked with input **⟨D⟩** we have pathological self-reference when D calls H with ⟨D⟩ and does the opposite of whatever H returns.

Does D halt on its own machine description ⟨D⟩ ?
This question can only be correctly answered after the pathology has been removed. When a halt decider only acts as a pure simulator of its input until after its halt status decision is made there is no feedback loop of back channel communication between the halt decider and its input that can prevent a correct halt status decision. In this case the halt decider is only examining the behavior of the input and has no behavior that can effect the behavior of this input thus can ignore it own behavior.

The standard pseudo-code halting problem template "proved" that the halting problem could never be solved on the basis that neither value of true (halting) nor false (not halting) could be correctly returned form the halt decider to the confounding input.

```
// Simplified Linz(1990) Ĥ and Strachey(1965) P
void P(u32 x)
{
  if (H(x, x))
    HERE: goto HERE;
}
```

This problem is overcome on the basis that a simulating halt decider would abort the simulation of its input before ever returning any value to this input. It aborts the simulation of its input on the basis that its input specifies what is essentially infinite recursion (infinitely nested simulation) to any simulating halt decider.

Every input to a simulating halt decider that only stops running when its simulation is aborted unequivocally specifies a computation that never halts. When input to a simulating halt decider cannot possibly reach its final state then we know that this input never halts.

A simulating halt decider H divides all of its input into:
(1) Those inputs that never halt unless H aborts their simulation (never halting).
    H aborts its simulation of these inputs an returns 0 for never halting.

(2) Those inputs that halt while H remains a pure simulator (halting).
    H waits for its simulation of this input to complete and then returns 1 halting.

Simulating partial halt decider H correctly decides that P(P) never halts

```
#include <stdint.h>
typedef void (*ptr)();

int H(ptr x, ptr y)
{
  x(y);
  return 1;
}

// Minimal essence of Linz(1990) Ĥ
// and Strachey(1965) P
void P(ptr x)
{
  H(x, x);
}

int main(void)
{
  H(P, P);
}
```

It is obvious that whether or not the above code is directly executed or H performs a pure simulation of its input that the above code specifies infinite recursion.

If H simulates its input in debug step mode it can correctly abort the simulation of this input as soon as H sees its simulated P call itself with the same parameters that it was called with. When it does this it correctly returns 0 for not halting.

```
_P()
[00001a5e](01)  55              push ebp
[00001a5f](02)  8bec            mov ebp,esp
[00001a61](03)  8b4508          mov eax,[ebp+08]
[00001a64](01)  50              push eax        // push P
[00001a65](03)  8b4d08          mov ecx,[ebp+08]
[00001a68](01)  51              push ecx        // push P
[00001a69](05)  e810000000      call 00001a7e   // call H
[00001a6e](03)  83c408          add esp,+08
[00001a71](01)  5d              pop ebp
[00001a72](01)  c3              ret
Size in bytes:(0021) [00001a72]
```

No computation halts (even if it stops running) unless it reaches its final state thus:
H(P,P) is only answering the question:
Does P ever reach its final state of 00001a72?

If H aborts the simulation of its input it does this at machine address: 00001a69, thus P never reaches 00001a72.

If H does not abort the simulation of its input H then infinite recursion continues to be invoked at machine address: 00001a69 and P never reaches 00001a72.

Turing machines must examine a (Turing machine description) mere representation of their input. H directly examines the actual machine language itself, thus no mere representation is involved. The actual infinitely recursive behavior of this input is easily verified by simply directly executing this machine language as shown above.

Peter Linz Ĥ applied to the Turing machine description of itself: ⟨Ĥ⟩

The following simplifies the syntax for the definition of the Linz Turing machine Ĥ, it is now a single machine with a single start state. A simulating halt decider is embedded at Ĥ.qx. It has been annotated so that it only shows Ĥ applied to ⟨Ĥ⟩, converting the variables to constants.

Ĥ.q0 ⟨Ĥ⟩ ⊢* Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞
If the UTM simulation of the input to Ĥ.qx ⟨Ĥ⟩ applied to ⟨Ĥ⟩ reaches its own final state.

Ĥ.q0 ⟨Ĥ⟩ ⊢* Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn
If the pure simulation of the input to Ĥqx ⟨Ĥ⟩ ⟨Ĥ⟩ would never reach its final state (whether or not this simulation is aborted) then it is necessarily true that Ĥqx transitions to Ĥ.qn correctly.

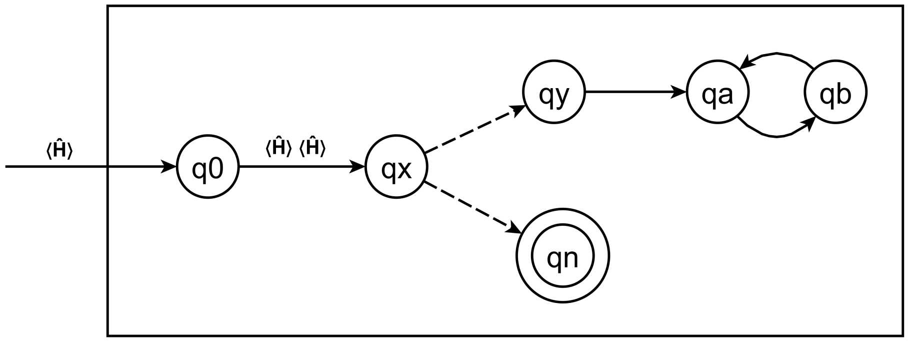

Figure 12.3 Turing Machine Ĥ applied to ⟨Ĥ⟩

Ĥ.q0 copies its input ⟨$Ĥ_0$⟩ to ⟨$Ĥ_1$⟩ then Ĥ.qx simulates this input $Ĥ_0$ with its input ⟨$Ĥ_1$⟩
$Ĥ_0$.q0 copies its input ⟨$Ĥ_1$⟩ to ⟨$Ĥ_2$⟩ then $Ĥ_0$.qx simulates this input $Ĥ_1$ with its input ⟨$Ĥ_2$⟩
$Ĥ_1$.q0 copies its input ⟨$Ĥ_2$⟩ to ⟨$Ĥ_3$⟩ then $Ĥ_1$.qx simulates this input $Ĥ_2$ with its input ⟨$Ĥ_3$⟩
$Ĥ_2$.q0 copies its input ⟨$Ĥ_3$⟩ to ⟨$Ĥ_4$⟩ then $Ĥ_2$.qx simulates this input $Ĥ_3$ with its input ⟨$Ĥ_4$⟩ ...

If the simulating halt decider at Ĥ.qx never aborts its simulation of its input this input never halts. If Ĥ.qx aborts its simulation of its input this input never reaches its final state and thus never halts. In all cases for every simulating halt decider at Ĥ.qx its input never halts.

When the pure simulation of the actual input to Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ never reaches the final state of this input then Ĥ.qx transitions to ⊢* Ĥ.qn is necessarily correct no matter what Ĥ ⟨Ĥ⟩ does. A halt decider is only accountable for correctly deciding the halt status of its actual input.

When the original Linz H is applied to ⟨Ĥ⟩ ⟨Ĥ⟩ it sees that its input transitions to Ĥ.qn. This provides the basis for H to transition to its final state of H.qy.

When Ĥ.qx is applied to ⟨Ĥ⟩ ⟨Ĥ⟩ it sees that none of the recursive simulations of its input ever halt it aborts the simulation of its input and correctly transitions to its final state of Ĥ.qn.

Strachey, C 1965. An impossible program The Computer Journal, Volume 7, Issue 4, January 1965, Page 313, https://doi.org/10.1093/comjnl/7.4.313

Linz, Peter 1990. An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (318-320)

Sipser, Michael 1997. Introduction to the Theory of Computation. Boston: PWS Publishing Company (165-167)